Ensembles et Séquences vs. Listes chaînées — Exercices corrigés

7 novembre 2018

Ce document montre des exemples de problèmes utilisant des ensembles et des séquences avec la représentation par liste chaînée. Il est conseillé de d'abord essayer de résoudre les exercices seul·e, puis dans un deuxième temps seulement de comparer votre solution avec la (ou les) solution(s) proposée(s).

Nous avons essayé pour chaque exercice de proposer une solution qui garantirait la note maximale à un·e étudiant·e. La solution est donc la plus concise et rapide possible.

Informations complémentaires

Vous trouverez dans ces encadrés des informations complémentaires qui ne seraient pas attendues dans une réponse d'étudiant·e, mais que nous trouvons important de mentionner pour vous aider dans votre apprentissage.

Les exercices suivants peuvent être résolus avec des tableaux ou des listes chaînées. Certains exercices étaient d'ailleurs présents dans la partie du cours sur la représentation par tableau avec longueur. Vous allez voir que les algorithmes haut niveau ne changent pas mais que les algorithmes de bas niveau sont eux différents.

Exercice 1 (Somme des éléments)

Écrire un algorithme haut-niveau puis un bas-niveau qui somme les éléments d'un ensemble d'entiers. En faire une analyse de complexité.

Exercice 2 (Recherche)

Écrire un algorithme haut-niveau puis un bas-niveau qui recherche si un élément appartient à un ensemble. L'algorithme haut-niveau renvoie un booléen, tandis que le bas-niveau pourra renvoyer la référence de la cellule de l'élément s'il a été trouvé, ou `Nil`

sinon. En faire une analyse de complexité.

Exercice 3 (Positifs/Négatifs)

Ecrire un algorithme haut-niveau puis un bas-niveau qui sépare une séquence en deux séquences. Si S est la séquence initiale, on veut garder les éléments positifs dans S et mettre les éléments négatifs dans une séquence $S2$. On veut conserver l'ordre initial et réaliser la séparation uniquement par modification des liens de chaînages (pas de suppression ni création de cellule, sauf éventuellement des cellules fictives).

Solution de l'exercice 1: Nous allons parcourir tous les éléments de l'ensemble et à rajouter les valeurs respectives à une variable `somme`. Cette variable est initialisée à 0. Notez que le résultat reste correct même si l'ensemble est vide.

Cas limites

Il est important de montrer que vous avez bien pensé aux cas « limites » (ou cas *particuliers*). Ici, quand l'ensemble est vide.

Algorithme haut niveau :

On fait pour l'instant abstraction de la représentation bas-niveau des ensembles.

```
somme_ensemble (E : Ensemble) : entier
    somme ← 0
    pour chaque e ∈ E faire
        somme ← somme + e
    retourner somme
```

Algorithme bas niveau : Avec une représentation par liste chaînée, si on utilise les types définis dans le poly du cours, cela donne :

```
somme_ensemble (S : Ensemble) : entier
    somme ← 0                          O(1)
    cel ← S.tete                       O(1)
    tant que cel ≠ Nil faire           n×
        somme ← somme + cel.valeur     O(1)
        cel ← cel.suivant              O(1)
    retourner somme                    O(1)
```

Notation algo versus notation langage C

Nous utilisons ici la notation du cours où pour accéder à l'élément référencé on écrit un '.'. Ne pas confondre avec la programmation en C où le '.' sert à accéder aux champs d'une structure et où pour accéder à un élément référencé il faut écrire '->'.

Analyse de complexité : L'initialisation (somme à 0) et la finalisation (retour de la valeur) sont en $O(1)$. Le corps de boucle est en $O(1)$ et la boucle est exécutée autant de fois que la longueur de l'ensemble. La complexité est donc de $O(1) + n \times O(1) + O(1) = O(n)$.

Solution de l'exercice 2: On initialise une variable booléenne `trouvé` avec la valeur `Faux` puis on parcourt un à un tous les éléments de l'ensemble. Si la valeur correspond à celle recherchée, on l'a trouvée et on met la variable `trouvé` à `Vrai`. Si la valeur recherchée n'est pas dans l'ensemble, la variable `trouvé` restera à `Faux`. À la fin du parcours on retourne la variable `trouvé`.

Cas particulier

On pourrait également préciser ici que si l'ensemble est vide, la valeur n'appartient pas à l'ensemble et donc la variable `trouvé` qui est à `Faux` est bien la valeur correcte à retourner.

Algorithme haut niveau

```
Rechercher(x : élément, E : Ensemble) : booléen
    trouvé ← Faux
    pour chaque e ∈ E faire
        si x = e alors
            trouvé ← Vrai
    retourner trouvé
```

Algorithme bas niveau

Dans le cas où un ensemble est représenté par liste chaînée, l'algorithme bas niveau peut retourner la référence de la cellule plutôt qu'un booléen. On utilise la valeur Nil si l'élément n'a pas été trouvé.

```
Rechercher(x : élément, E : Ensemble) : cellule
    trouvé ← Nil                          O(1)
    cel ← S.tete                          O(1)
    tant que cel ≠ Nil faire              n×
        si x = cel.valeur alors           O(1)
            trouvé ← cel                  O(1)
        cel ← cel.suivant                 O(1)
    retourner trouvé                      O(1)
```

Note : si l'ensemble contient plusieurs fois l'élément recherché, la cellule renvoyée sera la *dernière* occurrence.

Complexité : L'initialisation et la finalisation sont en $O(1)$. Le corps de boucle effectue une comparaison et parfois une affectation. Dans tous les cas, c'est un nombre constant d'opérations soit $O(1)$, et la boucle est exécutée n fois, avec n le nombre d'éléments de l'ensemble. L'algorithme a donc une complexité algorithmique en $O(n)$.

Algorithme plus efficace

Une solution plus efficace est d'arrêter le parcours dès que la valeur recherchée a été trouvée. (Ici, on pourrait même directement sortir de la fonction). Notez cependant que cela ne change pas la complexité (dans le pire des cas, l'élément recherché est le dernier et on doit tout de même parcourir tout l'ensemble). L'algorithme haut niveau devient :

```
Rechercher(x : élément, E : Ensemble) : booléen
    trouvé ← Faux
    pour chaque e ∈ E faire
        si x = e alors
            trouvé ← Vrai
            sortir de la boucle
        /* sinon, continuer                                  */
    retourner trouvé
```

Une implémentation bas-niveau possible, en utilisant des listes chaînées, une boucle "tant que", et en

renvoyant une cellule.

```
Rechercher(x : élément, E : Ensemble) : cellule
    trouvé ← Nil
    cel ← S.tete
    tant que cel ≠ Nil faire
        si x =cel.valeur alors
            trouvé ← cel
            break   /* quitte la boucle                                    */
        cel ← cel.suivant
    retourner trouvé
```

Notez que cette fois, s'il y a plusieurs fois l'élément cherché dans l'ensemble, on retourne l'indice de la *première* occurrence.

Solution de l'exercice 3: Principe de l'algorithme : je crée une séquence S_2 vide, puis je parcours S. Chaque fois que je rencontre une valeur négative, je la supprime de S et je l'ajoute en queue de S_2.

Algorithme haut-niveau :

```
Partition (S : référence de Séquence)
    S_2 ← nouvelle séquence
    pour chaque e ∈ S faire
        si e < 0 alors
            supprimer (S, e)
            ajouter (S_2, e)
```

En bas niveau, je vais avoir besoin dans le parcours de S d'une référence vers la cellule précédente (pour la suppression), et d'une référence vers la queue de S_2 (pour ajouter en queue sans tout reparcourir).

Pour simplifier les cas particuliers (problème de précédent sur la tête de S, et problème de queue quand S_2 est vide), je rajoute deux cellules fictives en tête de mes séquences.

Algorithme

```
Partition (S : référence de Séquence)
    S2 ← nouvelle séquence                                  } O(1)
    F ← nouvelle cellule
    F2 ← nouvelle cellule
    ajouter_debut (S, F)                                    /* cellules fictives */
    ajouter_debut (S2, F2)
    prec ← F
    queue ← F2
    cel ← F.suivant                                         /* cel est sur la première "vraie" cellule */
    tant que cel ≠ Nil faire                                } n×
        si cel.valeur < 0 alors                             } O(1)
            prec.suivant ← cel.suivant
            queue.suivant ← cel
            queue ← cel                                     /* Inutile de mettre queue.suivant à Nil ici */
        sinon
            prec ← cel                                      /* on se décale à la cellule suivante à tester */
        cel ← prec.suivant
    queue.suivant ← Nil                                     } O(1)
    supprime_tete(S)                                        /* Je supprime les cellules fictives */
    supprime_tete(S2)
    retourner S2
```

Attention, il est important, quand on déplace une cellule qui a une valeur négative, de **ne pas décaler "prec"** car on vient de changer son successeur : cette cellule n'a donc pas encore été testée.

Complexité

La complexité est en O(n) puisque la séparation se fait en un parcours de la séquence.